```
// LesAlgo.cpp : Voir aussi Lippman/Lajoie pp.1023-1097, Prata pp.802-812, 943-972, Stroustrup pp.507-5
// Par CG, MM mars 1998, rév. avril 1999, rév. février 2000
/* Il manque les choses de <numeric>...
adjacent_find  : recherche la première suite d'éléments identiques.
binary_search  : indique si une valeur est trouvée (renvoie un bool).
copy           : copie une séquence ailleurs.
copy_backward  : copie une séquence ailleurs (en partant de la fin).
count          : compte le nombre d'occurrences d'une valeur.
count_if       : compte le nombre d'éléments répondant à une condition.
equal          : compare deux séquences.
equal_range    : renvoie un intervalle contenant des copies de la donnée cherchée.
fill           : copie une valeur dans un intervalle.
fill_n         : copie plusieurs fois une valeur.
find           : recherche linéaire.
find_end       : recherche une séquence dans une autre, en partant de la fin.
find_first_of  : recherche le premier élément d'une séquence se trouvant dans une autre.
find_if        : recherche un élément répondant à une condition.
for_each        : passe chaque élément d'une séquence à une fonction.
generate         : remplit une séquence à partir des valeurs générées par une fonction.
generate_n        : crée une séquence de n valeurs générées par une fonction.
includes          : vérifie que toutes les valeurs d'une séquence sont dans une autre séquence.
inplace_merge      : fusionne une copie d'une séquence dans une autre séquence.
iter_swap           : échange les valeurs pointées par deux itérateurs.
lexicographical_compare : vérifie «lexicalement» deux séquences, élément par élément.
lower_bound          : recherche binaire de la première occurrence d'une valeur (ou plus grand).
make_heap           : convertit une séquence en «heap».
max                : renvoie une copie du maximum entre deux valeurs.
max_element       : renvoie la position du plus grand élément d'une séquence.
merge             : crée une nouvelle séquence contenant la fusion de copies de deux séquences.
min               : renvoie une copie du minimum entre deux valeurs.
min_element       : renvoie la position du plus petit élément d'une séquence.
mismatch          : recherche la position où les valeurs de deux séquences diffèrent.
next_permutation  : fait la prochaine permutation des éléments d'une séquence.
nth_element       : place le Nième élément d'une séquence et les plus petites avant lui.
partial_sort      : place à leur position les plus petits éléments d'une séquence.
partial_sort_copy : copie en ordre une certain nombre des plus petits éléments d'une séquence.
partition         : place les données répondant à une condition au début d'une séquence.
pop_heap          : enlève un élément et reconstruit le heap.
prev_permutation  : revient à la permutation précédente des éléments d'une séquence.
push_heap         : ajoute une valeur et reconstruit le heap.
random_shuffle  : mélange les données d'une séquence.
remove          : «enlève» toutes les copies d'une donnée dans une séquence.
remove_copy     : crée une copie des éléments différents d'une certaine valeur.
remove_copy_if  : crée une copie des éléments ne répondant pas à une condition.
remove_if       : «enlève» toutes les copies des données répondant à une condition.
replace         : remplace toutes les copies d'une valeur par une nouvelle valeur.
replace_copy    : crée une copie d'une séquence avec certaines données remplacées par une autre.
replace_copy_if : crée une copie avec certaines données remplacées par une autre.
replace_if      : remplace les valeurs répondant à une condition par une nouvelle valeur.
reverse         : renverse l'ordre des données.
reverse_copy    : crée une copie d'une séquence dans l'ordre inversée.
rotate          : transfère des données d'un bout de la séquence à l'autre bout.
rotate_copy     : fait une copie d'une séquence où on a transféré des données d'un bout de
                  la séquence à l'autre bout.
search          : recherche une séquence dans une autre, en partant du début.
search_n        : recherche une suite de N copies d'une valeur dans une séquence.
set_difference  : crée une séquence des éléments qui sont membres d'une séquence, mais
                  non membres d'une autre.
set_intersection       : crée une séquence des éléments appartenant aux deux séquences.
set_symmetric_difference : crée une séquence des éléments n'appartenant qu'à une de
                           deux séquences.
set_union        : crée une séquence contenant une copie (unique) des éléments de deux séquences.
sort             : trie une séquence.
sort_heap        : trie un séquence qui est déjà ordonnée en heap.
stable_partition : crée une partition prenant un temps stable.
stable_sort      : fait un tri prenant un temps stable.
swap             : échange le contenu de deux variables.
swap_ranges      : échange deux portions de séquence.
transform        : crée une séquence résultant de l'application d'une fonction sur les
                   données d'une ou deux séquences.
unique         : garde une seule copie des données d'une séquence ordonnée.
unique_copy    : crée une séquence avec une copie unique des données d'une séquence ordonnée.
upper_bound    : recherche binaire du premier élément plus grand qu'une valeur.
*/
```

```
#include <algorithm>
#include <iterator>
#include <functional>
#include <list>
#include <iostream>
#include <vector>
#include <string>
#include <stdcpp>
using namespace std;
/*
N.B. Les intervalles spécifiés sont toujours ouverts, c'est-à-dire que le dernier élément
     n'en fait pas partie [début, fin).

N.B. Plusieurs algorithmes existent en deux versions : la première fait ses comparaisons
     avec les opérateurs < ou == directement, alors que la deuxième permet de spécifier un
     prédicat (objet pouvant servir de condition, habituellement une fonction ou une
     fonction objet retournant un bool). Dans les exemples, on utilisera donc parfois :
*/
            bool MemeDizaine(int p_val1, int p_val2)
                {
                return p_val1/10 == p_val2/10;
                }

            bool EstPair(int p_val)
                {
                return p_val%2 == 0;
                }

            void EcrireDecHexOct(int p_val)
                {
                cout << p_val << " == 0x" << hex << uppercase << p_val
                     << " == 0" << oct << p_val << dec << endl;
                }

            struct ClEcrireSiDivisible : public binary_function<int, int, bool>
                {
                bool operator()(int p_val, int p_div) const
                    {
                    bool divisible= (p_div != 0 && p_val%p_div == 0);
                    cout << p_val <<  (divisible? " est" : " n'est pas")
                         << " divisible par " << p_div << endl;
                    return divisible;
                    }
                };

            int EcrireSi(int p_val, bool p_Pred(int))
                {
                if (p_Pred(p_val)) cout << p_val << endl;

                return 0; //...
                }
/*
ALGORITHMES NE TRAVAILLANT PAS NÉCESSAIREMENT SUR DES SÉQUENCES OU COLLECTION

    iter_swap : échange les valeurs pointées par deux itérateurs.
    max       : renvoie une copie du maximum entre deux valeurs.
    min       : renvoie une copie du minimum entre deux valeurs.
    swap      : échange le contenu de deux variables.
*/
int main()
    {
        int v[]= { 0, 1, 2, 5, 5, 5, 7, 9, 15, 17, 17 };
        const int NB_ELEM= sizeof(v)/sizeof(v[0]);

        vector<int> vtor(v, v+NB_ELEM);

        int valeur1= max(vtor.front(), vtor.back());
        int valeur2= min(*vtor.begin(), *(vtor.end()-1));
        int valeur3= max(valeur1, 99);

        swap(*vtor.begin(), *(vtor.end()-1));
        iter_swap(vtor.begin(), vtor.end()-1);  // Annule le précédent
        swap(valeur1, valeur3);
        cout << valeur1 << ' ' << valeur2 << ' ' << valeur3 << endl; // 99 0 17
```

```
/*
ALGORITHMES NE MODIFIANT PAS LA SÉQUENCE (SÉQUENCE == portion de collection) :

    Certains algorithmes demandent des «random iterator» et ne s'appliquent donc pas
    sur n'importe quel type de collections.

  Style recherches et vérification :

    adjacent_find : recherche la première suite d'éléments identiques.
    binary_search : indique si une valeur est trouvée (renvoie un bool).
    equal_range   : renvoie un intervalle contenant des copies de la donnée cherchée.
    find          : recherche linéaire.
    find_end      : recherche une séquence dans une autre, en partant de la fin.
    find_first_of : recherche le premier élément d'une séquence se trouvant dans une autre.
    find_if       : recherche un élément répondant à une condition.
    includes      : vérifie que toutes les valeurs d'une séquence sont dans une autre séquence.
    lexicographical_compare : vérifie «lexicalement» deux séquences, élément par élément.
    lower_bound   : recherche binaire de la première occurrence d'une valeur (ou plus grand).
    max_element   : renvoie la position du plus grand élément d'une séquence.
    min_element   : renvoie la position du plus petit élément d'une séquence.
    search        : recherche une séquence dans une autre, en partant du début.
    search_n      : recherche une suite de N copies d'une valeur dans une séquence.
    upper_bound   : recherche binaire du premier élément plus grand qu'une valeur.
*/
// Rappel : v[]= { 0, 1, 2, 5, 5, 5, 7, 9, 15, 17, 17 };
        list<int> liste(v, v+NB_ELEM);

        int* itv= adjacent_find(v, v+NB_ELEM); //itv == &v[3] i.e. le premier 5
        if (itv == v+NB_ELEM) cout << "Erreur!";

        if (binary_search(itv+1, v+NB_ELEM, *itv+10))
            cout << "Normal, 15 est dans le vecteur\n";

        list<int>::iterator itl= find(liste.begin(), liste.end(), 9);
        if (liste.end() != itl)
            {
            list<int>::iterator it= itl;                            // On ne peut pas
            cout << "On a trouvé 9 juste après " << *--it << endl;  //  faire itl-1...
            }

        itl= find_end(liste.begin(), liste.end(), v+1, v+4); // *itl==1
        itl= find_end(liste.begin(), liste.end(), v+1, v+4, MemeDizaine); // *itl==dernier 5

        int v512[3]= {5, 1, 2};
        itl= find_first_of(liste.begin(), liste.end(), v512, v512+3); // *itl==1
        itv= find_if(v, v+NB_ELEM, bind2nd(greater<int>(), 10));      // *itv==15

        if (!includes(v+2, v+4, v+2, v+6)) cout << "bug!";
        if (includes(v+2, v+4, v+2, v+7)) cout << "bug!";

        if (lexicographical_compare(v+3, v+6, v512, v512+3)) cout << "bug 5 5 5 < 5 1 2 !";
        if (lexicographical_compare(v512, v512+1, v512, v512+2))
            cout << "Normal 5 < (5 1) !\n";

        if (lower_bound(v, v+NB_ELEM, 4) == upper_bound(v, v+NB_ELEM, 4)
            && lower_bound(v, v+NB_ELEM, 5) != upper_bound(v, v+NB_ELEM, 5)
            && equal_range(v, v+NB_ELEM, 5).second == equal_range(v, v+NB_ELEM, 7).first)
            cout << "On comprend tout!\n";

        cout << "Plus grand= " << *max_element(liste.begin(), liste.end()) << endl
             << "Ou encore= " << *min_element(v, v+NB_ELEM, greater<int>()) << endl;

        if (v+NB_ELEM != search(v, v+NB_ELEM, v512, v512+3)) cout << "bug 512 dans v?";
        if (liste.end() == search_n(liste.begin(), liste.end(), 3, 5)) cout << "bug 5 5 5";
        if (liste.end() != search_n(liste.begin(), liste.end(), 3, 12, MemeDizaine))
            cout << "Normal il y a trois valeurs de la même dizaine que 12 (c-à-d 15 17 17)\n";
```

```
/*
  Comparaisons :
    equal    : compare deux séquences.
    mismatch : recherche la position où les valeurs de deux séquences diffèrent.

  Comptage :
    count    : compte le nombre d'occurrences d'une valeur.
    count_if : compte le nombre d'éléments répondant à une condition.

  Général :
    for_each : passe chaque élément d'une séquence à une fonction. La fonction ne devrait pas
               changer les données, utiliser transform (plus explicite) quand c'est le cas.
*/
        if (!equal(liste.begin(), liste.end(), v)) cout << "Erreur, ont les mêmes données.";
        if (equal(v+3, v+4, v512))
            cout << "Normal car regarde pas plus loin que ce qu'on demande.\n";

        cout << "5 5 5 diffère de 5 5 7 par le dernier " << *mismatch(v+3, v+6, v+4).first
             << " qui est différent du " << *mismatch(v+3, v+6, v+4).second << endl;
        pair<int*, int*> pos= mismatch(v+3, v+5, v+3);
        if (pos.first != pos.second || pos.first != v+5) cout << "bug 5 5 != 5 5";

        cout << "On le sait, il y a " << count(v, v+NB_ELEM, 5) << " 5 dans v.\n"
             << "Il y a aussi "
             << count_if(v, v+NB_ELEM, ptr_fun(EstPair)) << " nombres pairs et "
             << count_if(v, v+NB_ELEM, not1(bind2nd(ptr_fun(MemeDizaine), 12)))
             << " d'une dizaine différente de celle de 12\n";

        for_each(liste.begin(), liste.end(), EcrireDecHexOct);
        for_each(liste.begin(), liste.end(), bind2nd(ClEcrireSiDivisible(), 2));
/*
ALGORITHMES MODIFIANT LA SÉQUENCE

  Général :
    transform : crée une séquence résultant de l'application d'une fonction sur les
                données d'une ou deux séquences.
*/
        transform(v, v+NB_ELEM, v, bind2nd(plus<int>(), 20)); // +20 à chaque élément
    //  int v[]= {  0,  1,  2,  5,  5,  5,  7,  9, 15, 17, 17 };
    //  +          20  20  20  20  20  20  20  20  20  20  20
    //             --  --  --  --  --  --  --  --  --  --  --
    //           { 20, 21, 22, 25, 25, 25, 27, 29, 35, 37, 37 }
        transform(v, v+NB_ELEM-2, v+2, v, plus<int>()); // v[i]= v[i]+v[i+2];
    //           { 20, 21, 22, 25, 25, 25, 27, 29, 35, 37, 37 }
    //  +  XX, XX, 22, 25, 25, 25, 27, 29, 35, 37, 37
    //             --  --  --  --  --  --  --  --  --
    //           { 42, 46, 47, 50, 52, 54, 62, 66, 72, 37, 37 }
        transform(v+NB_ELEM-2, v+NB_ELEM, v+NB_ELEM-2, bind2nd(multiplies<int>(), 2));
    //           { 42, 46, 47, 50, 52, 54, 62, 66, 72, 37, 37 }
    //                                                 x2  x2
    //                                                 --  --
    //           { 42, 46, 47, 50, 52, 54, 62, 66, 72, 74, 74 }
/*
  Copie : (si on copie dans la même séquence, il faut bien choisir entre copy et backward_copy)
    copy          : copie une séquence ailleurs.
    copy_backward : copie une séquence ailleurs (en partant de la fin).
*/
        copy(v, v+NB_ELEM, ostream_iterator<int>(cout, " "));   // Affichage...
        cout << endl;
        // On a donc pour le moment :
        //     0   1   2   3   4   5   6   7   8   9  10
        //  { 42, 46, 47, 50, 52, 54, 62, 66, 72, 74, 74 }

        copy(v+7, v+10, v+8); // Il y a un paramètre déduit...
        // copy(v+7, v+10, v+8 /*, v+11*/);  // Copier [7,10) vers [8,11)
        // On dirait
        //  { 42, 46, 47, 50, 52, 54, 62, 66, 72, 74, 74 }
        //                  Ceci :          a   b   c
        //                  à copier là :       A   B   C
        //
        // Mais d'abord a dans A :          a   A
        //  { 42, 46, 47, 50, 52, 54, 62, 66, 66, 74, 74 }
        //
        // b dans B ensuite                     b   B
        //  { 42, 46, 47, 50, 52, 54, 62, 66, 66, 66, 74 }
        //
        // puis c dans C                            c   C
        //  { 42, 46, 47, 50, 52, 54, 62, 66, 66, 66, 66 }
```

```
        copy(v, v+NB_ELEM, ostream_iterator<int>(cout, ", "));
        cout << endl;
        // 42, 46, 47, 50, 52, 54, 62, 66, 66, 66, 66,

        copy_backward(v+2, v+5 /*,v+3, */, v+6);    // Copier [2,5) vers [3,6)
        //  { 42, 46, 47, 50, 52, 54, 62, 66, 66, 66, 66 }
        //              a   b   c
        //                  A   B   C
        // D'abord cC :
        //  { 42, 46, 47, 50, 52, 52, 62, 66, 66, 66, 66 }
        // Ensuite bB :
        //  { 42, 46, 47, 50, 50, 52, 62, 66, 66, 66, 66 }
        // Puis aA :
        //  { 42, 46, 47, 47, 50, 52, 62, 66, 66, 66, 66 }
        copy(v, v+NB_ELEM, ostream_iterator<int>(cout, ", "));
        cout << "\b\b \n";
        // 42, 46, 47, 47, 50, 52, 62, 66, 66, 66, 66

// Pour simplifier, on utilisera cette macro (on ne pouvait pas déclarer une fonction ici)
#define Afficher(debut, fin)    copy(debut, fin, ostream_iterator<int>(cout, ", ")); \
                                cout << "\b\b \n";
/*
  Remplissage :
    fill       : copie une valeur dans un intervalle
    fill_n     : copie plusieurs fois une valeur
    generate   : remplit une séquence à partir des valeurs générées par une fonction
    generate_n : crée une séquence de n valeurs générées par une fonction
*/
        fill(v, v+3, 0);                    // 0, 0, 0, 47, 50, 52, 62, 66, 66, 66, 66
        fill_n(v+3, 3, 1);                  // 0, 0, 0, 1, 1, 1, 62, 66, 66, 66, 66
        generate(v+6, v+7, rand);           // 0, 0, 0, 1, 1, 1, ?, 66, 66, 66, 66
        generate_n(v+7, NB_ELEM-7, rand);   // 0, 0, 0, 1, 1, 1, ?, ??, ??, ??, ??
        Afficher(v, v+NB_ELEM);
        // 0, 0, 0, 1, 1, 1, 41, 18467, 6334, 26500, 19169
/*
  Tri et fusion :
    inplace_merge     : fusionne une copie d'une séquence dans une autre séquence.
    merge             : crée une séquence contenant la fusion de copies de deux séquences.
    nth_element       : trouve et place le Nième élément d'une séquence avec les valeurs plus
                        petites avant lui.
    partial_sort      : place à leur position les plus petits éléments d'une séquence.
    partial_sort_copy : crée une copie ordonnée d'une certain nombre des plus petits éléments
                        d'une séquence.
    partition         : réordonne les données d'une séquence pour que les données répondant
                        à une condition soient au début.
    sort :              trie une séquence.
    stable_partition  : crée une partition prenant un temps stable.
    stable_sort       : fait un tri prenant un temps stable.

  Mélange et permutation :
    next_permutation  : fait la prochaine permutation des éléments d'une séquence.
    prev_permutation  : revient à la permutation précédente des éléments d'une séquence.
    random_shuffle    : mélange les données d'une séquence.
    reverse           : renverse l'ordre des données.
    reverse_copy      : crée une copie d'une séquence dans l'ordre inversée.
    rotate            : transfère des données d'un bout de la séquence à l'autre bout.
    rotate_copy       : fait une copie d'une séquence où on a transféré des données d'un bout
                        de la séquence à l'autre bout.
    swap_ranges       : échange deux portions de séquence.
*/
        vector<int> vi(v, v+NB_ELEM);
        random_shuffle(vi.begin(), vi.end());
        Afficher(vi.begin(), vi.end());     // 1, 19169, 1, 0, 1, 18467, 26500, 41, 0, 0, 6334
        sort(vi.begin()+1, vi.begin()+4);
        Afficher(vi.begin(), vi.end());     // 1, 0, 1, 19169, 1, 18467, 26500, 41, 0, 0, 6334
        partial_sort(vi.begin(), vi.begin()+3, vi.end());  // On veut les 3 plus petits
        Afficher(vi.begin(), vi.end());     // 0, 0, 0, 19169, 1, 18467, 26500, 41, 1, 1, 6334
/*  partial_sort serait plus clair ainsi

    template <typename It>
    inline void PlacerLesNPlusPetits(int n, It debut, It fin)
            {
            partial_sort(debut, debut+n, fin);
            }

    on pourrait alors faire :   PlacerLesNPlusPetits(3, vi.begin(), vi.end());
*/
```

```
        nth_element(vi.begin(), vi.begin()+6, vi.end());  // on veut le bon 6ème
        // x, x, x, x, x, x, 41, x, x, x, x

        random_shuffle(vi.begin(), vi.end());
        vector<int>::iterator fin= partition(vi.begin(), vi.end(), EstPair);
        Afficher(vi.begin(), vi.end());
        // 6334, 0, 26500, 0, 0, 41, 18467, 1, 1, 1, 19169
        //                        fin

        fin= partition(vi.begin(), vi.end(), bind2nd(greater<int>(),10000));
        Afficher(vi.begin(), vi.end());
        // 19169, 18467, 26500, 0, 0, 41, 0, 1, 1, 1, 6334
        //                      fin

        reverse(vi.begin(), vi.end());
        Afficher(vi.begin(), vi.end());
        // 6334, 1, 1, 1, 0, 41, 0, 0, 26500, 18467, 19169

        rotate(vi.begin(), vi.begin()+3, vi.end());
        Afficher(vi.begin(), vi.end());
        // 1, 0, 41, 0, 0, 26500, 18467, 19169, 6334, 1, 1
        swap_ranges(vi.begin(), vi.begin()+3, vi.begin()+3);
        Afficher(vi.begin(), vi.end());
        // 0, 0, 26500, 1, 0, 41, 18467, 19169, 6334, 1, 1

        sort(v, v+NB_ELEM);             // Par défaut, comme sort(v, v+NB_ELEM, less<int>());
        sort(v, v+NB_ELEM, greater<int>());
        sort(vi.begin(), vi.end(), greater<int>());
        Afficher(vi.begin(), vi.end());
        // 26500, 19169, 18467, 6334, 41, 1, 1, 1, 0, 0, 0  // v est identique...

        vector<int> resultat(NB_ELEM+vi.size());    // On réserve assez de place
        merge(v, v+NB_ELEM, vi.begin(), vi.end(), resultat.begin(), greater<int>());
        Afficher(resultat.begin(), resultat.end());
        // 26500, 26500, 19169, 19169, 18467, 18467, 6334, ...

        list<int> fusion(v, v+NB_ELEM);
        list<int>::iterator milieu= fusion.begin();
        fusion.insert(fusion.begin(), vi.begin(), vi.end());
        inplace_merge(fusion.begin(), milieu, fusion.end(), greater<int>());
        Afficher(fusion.begin(), fusion.end());
        // 26500, 26500, 19169, 19169, 18467, 18467, 6334, ...

        reverse_copy(fusion.begin(), fusion.end(), resultat.begin()); // fusion ne change pas
        Afficher(resultat.begin(), resultat.end());
        // 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 41, 41, 6334, 6334, 18467, ...

    /* N.B. Les algorithmes de style :  XXX_copy_YYY(..., src, ... dest, ...)

            font l'équivalent de :      copy(src, dest, ...); //ce N'EST PAS un insert
                                        XXX_YYY(dest);

            Ex. reverse_copy(...) est un copy(...) puis une reverse(...) (ou l'équivalent)
                (il n'y a pas de YYY ici mais on verra plus loin des replace_copy_if, etc.)
     */
        v[0]= 0; v[1]= 1; v[2]=2; v[3]= 3;
        for (int i= 0; i < 6; ++i)
            {
            next_permutation(v, v+4);
            cout << v[0] << v[1] << v[2] << v[3] << endl;
            }
        // (0123) 0132  0213  0231  0312  0321  1023 (...)
```

```
/*
  Modification des éléments :
    remove           : enlève toutes les copies d'une donnée dans une séquence.
    remove_copy      : crée une copie des éléments différents d'une certaine valeur.
                       (en principe, enlève les éléments dans la copie de la séquence)
    remove_copy_if   : crée une copie des éléments ne répondant pas à une condition.
    remove_if        : enlève toutes les copies des données répondant à une condition.
    replace          : remplace toutes les copies d'une valeur par une nouvelle valeur.
    replace_copy     : crée une copie d'une séquence avec certaines données
                       remplacées par une autre.
    replace_copy_if  : crée une copie d'une séquence avec certaines données remplacées.
    replace_if       : remplace toutes les copies des valeurs répondant à une condition.
    unique           : garde une seule copie des données d'une séquence ordonnée.
    unique_copy      : crée une séquence contenant une seule copie des données d'une
                       séquence ordonnée
*/
        resultat.erase(resultat.begin()+17, resultat.end());
        // resultat au départ == 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 41, 41, 6334, 6334, 18467

        fin= remove(resultat.begin(), resultat.end(), 1);
        Afficher(resultat.begin(), resultat.end());
        // 0, 0, 0, 0, 0, 0, 41, 41, 6334, 6334, 18467, 1, 41, 41, 6334, 6334, 18467
        //                                              fin
        resultat.erase(fin, resultat.end());
        Afficher(resultat.begin(), resultat.end());
        // 0, 0, 0, 0, 0, 0, 41, 41, 6334, 6334, 18467

        fusion.erase(remove_copy(resultat.begin(), resultat.end(), fusion.begin(), 0),
                     fusion.end());
        Afficher(fusion.begin(), fusion.end());
        // 41, 41, 6334, 6334, 18467

        replace(fusion.begin(), fusion.end(), 41, 77);
        replace_if(fusion.begin(), fusion.end(), EstPair, 100);
        replace_if(fusion.begin(), fusion.end(), bind2nd(greater<int>(), 15000), 999);
        Afficher(fusion.begin(), fusion.end());
        // 77, 77, 100, 100, 999

        fin= unique_copy(fusion.begin(), fusion.end(), resultat.begin());
        // 77, 100, 999, 0, 0, 0, 41, 41, 6334, 6334, 18467
        //               fin
        resultat.erase(fin, resultat.end());
        Afficher(resultat.begin(), resultat.end());
        // 77, 100, 999
        // fusion.erase(unique(fusion.begin(), fusion.end()), fusion.end()); // Marcherait...
        // fusion.unique(); // Est plus efficace... et plus simple.
```

```
/*  Gestion d'éléments style «ensemble»
    N.B.: Les séquences doivent être ordonnées et ne pas contenir de duplicatas (sinon...
          faites vos propres tests !).

    includes         : vérifie si chaque membre de la 1ère séquence est aussi un membre de
                       la 2ième (voir aussi plus haut).
    set_difference   : crée une séquence des éléments qui sont membres de la 1ère séquence
                       mais non membres de la seconde (l'aide en ligne est erronée).
    set_intersection : crée une séquence des éléments appartenant aux deux séquences.
    set_symmetric_difference : crée une séquence des éléments n'appartenant qu'à une de
                               deux séquences.
    set_union        : crée une séquence contenant une copie (unique) des éléments de
                       deux séquences.
*/
    list<int> lset1;
    lset1.push_back(10);lset1.push_back(15);lset1.push_back(25);
    list<int> lset2(lset1.begin(), lset1.end());
    lset2.push_front(5); lset2.push_front(3); lset2.push_front(1);lset2.push_back(35);
    lset1.push_back(30);
    Afficher(lset1.begin(), lset1.end());   // 10, 15, 25, 30
    Afficher(lset2.begin(), lset2.end());   // 1, 3, 5, 10, 15, 25, 35

    list<int> vset(50,0);   // Faut réserver de la place pour la séquence destination
    list<int>::iterator it; // Recevra la position de fin de l'opération

    it= set_difference(lset1.begin(), lset1.end(), lset2.begin(),lset2.end(), vset.begin());
    Afficher(vset.begin(), it);
    //30
    it= set_difference(lset2.begin(), lset2.end(), lset1.begin(),lset1.end(), vset.begin());
    Afficher(vset.begin(), it);
    //1, 3, 5, 35
    it= set_intersection(lset1.begin(), lset1.end(), lset2.begin(),lset2.end(), vset.begin());
    Afficher(vset.begin(), it);
    //10, 15, 25
    it= set_symmetric_difference(lset1.begin(), lset1.end(), lset2.begin(),lset2.end(),
                                 vset.begin());
    Afficher(vset.begin(), it);
    //1, 3, 5, 30, 35
    it= set_union(lset1.begin(), lset1.end(), lset2.begin(),lset2.end(), vset.begin());
    Afficher(vset.begin(), it);
    //1, 3, 5, 10, 15, 25, 30, 35

    reverse(lset1.begin(), lset1.end()); reverse(lset2.begin(), lset2.end());
    it= set_union(lset1.begin(), lset1.end(), lset2.begin(),lset2.end(), vset.begin(),
        greater<int>());
    Afficher(vset.begin(), it);
    //35, 30, 25, 15, 10, 5, 3, 1

/*  Gestion de heap :
    make_heap : convertit une séquence en «heap» (monceau).
    pop_heap  : enlève un élément (le met à la fin) et reconstruit le heap.
    push_heap : ajoute une valeur (qu'on a mise à la fin) et reconstruit le heap.
    sort_heap : trie un séquence qui est déjà ordonnée en heap.
*/
    vector<int> vh;
    for (int i= 0; i<10; ++i) vh.push_back(rand()%30);
    Afficher(vh.begin(), vh.end());     //22, 21, 26, 8, 5, 17, 6, 11, 18, 9

    make_heap(vh.begin(), vh.end());        //               26
    Afficher(vh.begin(), vh.end());         //             /    \
    //26, 21, 22, 18, 9, 17, 6, 11, 8, 5    //           21      22
                                            //          /  \    /  \
                                            //         18   9  17   6
                                            //        / \   /
                                            //       11  8 5

    pop_heap(vh.begin(),vh.end());      // Monte le 5 en haut et le redescend en échangeant
    Afficher(vh.begin(), vh.end());     //  avec le plus grand fils...
    //22, 21, 17, 18, 9, 5, 6, 11, 8, 26

    push_heap(vh.begin(),vh.end());     // Insère au bas (déjà là) et échange si père est
    Afficher(vh.begin(), vh.end());     //  plus petit...
    //26, 22, 17, 18, 21, 5, 6, 11, 8, 9

    sort_heap(vh.begin(),vh.end());     // Répète pop_heap... pour finir le tri
    Afficher(vh.begin(), vh.end());
    //5, 6, 8, 9, 11, 17, 18, 21, 22, 26
}
```

```
/******* RÉSULTATS *******
99 0 17
Normal, 15 est dans le vecteur
On a trouvé 9 juste après 7
Normal 5 < (5 1) !
On comprend tout!
Plus grand= 17
Ou encore= 17
Normal il y a trois valeurs de la même dizaine que 12 (c-à-d 15 17 17)
Normal car regarde pas plus loin que ce qu'on demande.
5 5 5 diffère de 5 5 7 par le dernier 5 qui est différent du 7
On le sait, il y a 3 5 dans v.
Il y a aussi 2 nombres pairs et 8 d'une dizaine différente de celle de 12
0 == 0x0 == 00
1 == 0x1 == 01
2 == 0x2 == 02
5 == 0x5 == 05
5 == 0x5 == 05
5 == 0x5 == 05
7 == 0x7 == 07
9 == 0x9 == 011
15 == 0xF == 017
17 == 0x11 == 021
17 == 0x11 == 021
0 est divisible par 2
1 n'est pas divisible par 2
2 est divisible par 2
5 n'est pas divisible par 2
5 n'est pas divisible par 2
5 n'est pas divisible par 2
7 n'est pas divisible par 2
9 n'est pas divisible par 2
15 n'est pas divisible par 2
17 n'est pas divisible par 2
17 n'est pas divisible par 2
42 46 47 50 52 54 62 66 72 74 74
42, 46, 47, 50, 52, 54, 62, 66, 66, 66, 66,
42, 46, 47, 47, 50, 52, 62, 66, 66, 66, 66
0, 0, 0, 1, 1, 1, 41, 18467, 6334, 26500, 19169
1, 19169, 1, 0, 1, 18467, 26500, 41, 0, 0, 6334
1, 0, 1, 19169, 1, 18467, 26500, 41, 0, 0, 6334
0, 0, 0, 19169, 1, 18467, 26500, 41, 1, 1, 6334
6334, 0, 26500, 0, 0, 41, 18467, 1, 1, 1, 19169
19169, 18467, 26500, 0, 0, 41, 0, 1, 1, 1, 6334
6334, 1, 1, 1, 0, 41, 0, 0, 26500, 18467, 19169
1, 0, 41, 0, 0, 26500, 18467, 19169, 6334, 1, 1
0, 0, 26500, 1, 0, 41, 18467, 19169, 6334, 1, 1
26500, 19169, 18467, 6334, 41, 1, 1, 1, 0, 0, 0
26500, 26500, 19169, 19169, 18467, 18467, 6334, 6334, 41, 41, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0
26500, 26500, 19169, 19169, 18467, 18467, 6334, 6334, 41, 41, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 41, 41, 6334, 6334, 18467, 18467, 19169, 19169, 26500, 26500
0132
0213
0231
0312
0321
1023
0, 0, 0, 0, 0, 0, 41, 41, 6334, 6334, 18467, 1, 41, 41, 6334, 6334, 18467
0, 0, 0, 0, 0, 0, 41, 41, 6334, 6334, 18467
41, 41, 6334, 6334, 18467
77, 77, 100, 100, 999
77, 100, 999
10, 15, 25, 30
1, 3, 5, 10, 15, 25, 35
30
1, 3, 5, 35
10, 15, 25
1, 3, 5, 30, 35
1, 3, 5, 10, 15, 25, 30, 35
35, 30, 25, 15, 10, 5, 3, 1
22, 21, 26, 8, 5, 17, 6, 11, 18, 9
26, 21, 22, 18, 9, 17, 6, 11, 8, 5
22, 21, 17, 18, 9, 5, 6, 11, 8, 26
26, 22, 17, 18, 21, 5, 6, 11, 8, 9
5, 6, 8, 9, 11, 17, 18, 21, 22, 26
*/
```